REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1989	Final Technical 2/1/89 - 8/1/89

4. TITLE AND SUBTITLE
A Genetic Adaptive System for Image Understanding and Learning Research

5. FUNDING NUMBERS
F49620-89-C-0039

6. AUTHOR(S)
Dean Z. Douthat / Kevin W. Ross

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Institute for the Study of Intelligent Systems
P.O. Box 669
Ann Arbor, Michigan 48105-0669

8. PERFORMING ORGANIZATION REPORT NUMBER
I0037U/89-02C

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Sponsor: DARPA/TTO 1400 Wilson Blvd. Arlington Va 22209
Monitor: Air Force Office of Scientific Research/NE
Bldg 410, Bolling AFB DC 20332-6448

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFOSR-TR-89-1391

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report documents the efforts and results of initial phase research on machine learning directed at application for real time machine vision and automatic target recognition. The particular paradigm pursued is based on genetic algorithms and classifiers modeled on the summation of Mendelian genetic recombination, Darwinian selection and ecological notions of competition. This machine learning approach is strongly supported by sound statistical theory.

A second thread of research was the development of massively parallel computing hardware based on the Geometric/Arithmetic Parallel Processor [GAPP]. This machine has a large number of processors, each one bit wide with a full Arithmetic/Logic Unit [full adder] and with local memory per processor. The basic research hypothesis of the subject effort has been that GAPP contained sufficient hardware capability to provide a substrate for a Classifier and Genetic Algorithm system. The goal has been demonstrated by constructing and running the necessary software on the GAPP, its controller and its host. The resulting fusion of software and hardware is called a Genetic Algorithm/Classifier Engine [GACE] in the same sense as a LISP engine or a database engine. The resulting quantum jump in performance should open doors both to application and to more interesting and relevant research.

14. SUBJECT TERMS	15. NUMBER OF PAGES
	26
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

NSN 7540-01-280-5500

Standard Form 298 (890104 Draft)
Prescribed by ANSI Std. 239-18
298-01

"Original contains color plates: All DTIC reproductions will be in black and white."

ISIS NUMBER I0037U/89-02C

A Genetic Adaptive System For Image Understanding And Learning Research

Phase I Final Report
August 1989

Sponsored by:

DEFENSE ADVANCED RESEARCH PROJECTS AGENCY
DARPA ORDER NO. 6557

Monitored by:

USAF, AFSC
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH
BUILDING 410
BOLLING AFB, D.C. 20332-6448
UNDER CONTRACT NO. F49620-89-C-0039

Prepared by:

INSTITUTE FOR THE STUDY OF INTELLIGENT SYSTEMS
P.O. BOX 669
ANN ARBOR, MICHIGAN 48105

Contents

Figures

Section 1

Summary

This report documents the efforts and results of the initial phase research on machine learning ultimately aimed at application for real time machine vision and automatic target recognition. The particular paradigm being pursued is based on genetic algorithms and classifiers invented and developed by Prof. John Holland of the University of Michigan during a period approaching two decades. They are modeled on the powerful and successful natural learning system represented by the summation of Mendelian genetic recombination, Darwinian selection and ecological notions of competition -- the "Modern Synthesis".[1] This machine learning approach is strongly supported by sound statistical theory.

Application of this approach has not taken place. The major impediment is that genetic algorithms and classifiers share with all statistical methods the need for large populations or samples for efficiency and efficacy. For serial computer implementations, this translates directly into running time increases on the order of the square of population. In practice, this characteristic has precluded application to real world-sized problems with real time constraints.

A second thread of research has been the development of massively parallel computing hardware for image processing. Of particular interest is the Geometric/Arithmetic Parallel Processor [GAPP] developed by Martin Marietta of Orlando FL. This machine has a large number of processors, each one bit wide with a full Arithmetic/Logic Unit [full adder] and with local memory per processor. The basic research hypothesis of the subject effort has been that GAPP contained sufficient hardware capability to provide a substrate for a genetic algorithm and classifier system. The goal was to demonstrate this by constructing and running the necessary software on the GAPP, its controller and its host. We call the resulting fusion of software and hardware a Genetic Algorithm/Classifier Engine [GACE] in the same sense as a LISP engine or a database engine. The resulting quantum jump in performance should open doors both to application and to more interesting and relevant research.

As the remainder of this report shows, the hypothesis has been proven true and a GACE is now in operation. In the following sections, we discuss hardware, software and test results. We have built the classifier system using as a model the Classifier System in C [CFS-C] program written by Rick Riolo which is the *de facto* standard. We acknowledge considerable assistance and helpful conversations with Dr. Riolo in understanding his software, in addressing design issues for our software and in anticipating problems.

Additionally, we happily acknowledge the help of NCR Corporation who supplied a GAPP machine for use in this research along with host machines and continuing support. In particular, Lee Hoevel of NCR Corporate Headquarters in Dayton OH, Dave Ruhberg of NCR Microelectronics Division in Fort Collins CO, and Peter Robinson of NCR CANADA LTD, Engineering and Manufacturing Division, Waterloo ON, have given us invaluable material, technical and moral support.

---

1. Booker, L. B., D. E. Goldberg & J. H. Holland "Classifier Systems and Genetic Algorithms" Technical Report #8, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, Ann Arbor MI 1987

Section 2

Hardware

This section expands and supplements preliminary NCR SPDS documentation.[2] Besides completing and adding detail, it relates hardware features to programming requirements for accessing and using these features. In its preliminary version, the hardware manual alone does not suffice for understanding operation and programming at low levels. The understanding reflected herein was gleaned through study of schematics and existing software. No NCR proprietary software, methods or trade secrets are disclosed herein.

2.1 OVERVIEW

The hardware is designed to allow arrays of GAPP VLSI chips [NCR part NCR45CG72] to be controlled and used by a PC/AT compatible host computer. Each chip contains 72 processors in a 12X6 array with mesh connection topology all executing the same instruction stream on data in memory local to each processor. This Single Instruction/Multiple Data stream [SIMD] arrangement is continued by chip-to-chip connections extending the mesh connectivity. GAPP chips are mounted on pc boards in five rows of eight chips each giving an array 60 high by 48 wide [2880]. Jumpers allow this array to be treated monolithically as a 60X48 mesh or to isolate the bottom row from the mesh and make it a corner turner.

In the latter arrangement, the top row of processors in the bottom row of eight chips [12X48] is not connected via North-South [ns] data registers to the bottom row of processors in the next chip row up. The sole upward data path is the separate communications [cm] register which only connects so as to allow South to North data flow. Also, the bottom chip row has its instruction stream isolated from the other four resulting in a Dual Instruction/Multiple Data system. With this arrangement, serial or raster data can be shifted into the bottom row, for example from East to West and then a whole row sent Northward in parallel. The cm registers of the top row of processors are connected back to the bottom row of processors of the corner turn [CT] array so that output data return to the corner turn as new data flow in. These output data shift out the West end of the corner turn array during shift in of the next raster input line. Thus both the serial [shift] and parallel [take over] phases can be made systolic and coordinated with the main array [MA] processing.

Up to four GAPP boards may be used in a single SPDS system, arranged as "lower" boards with CT and MA or "upper" boards with MA only. Practical configurations are:

Boards	Upper	Lower	CT	MA	MA Processors
4	2	2	12X96	108X96	10368
2	1	1	12X48	108X48	5184
1	0	1	12X48	48X48	2304

In any case, the operation of GAPP boards is controlled by a pc board in the SPDS housing which provides the two instruction streams to CT and MA as well as facilities for transferring data to and from the host. These transfers occur via a Universal Host Interface [UHI] card mounted in an AT compatible [16 bit] host slot. The UHI provides signals for both Programmed Input/Output [PIO] and Direct Memory Access [DMA] transactions with the host. Thus there are four major functions:

---

2. "SIMD Development System Hardware Manual", NCR Microelectronics - Fort Collins CO, Rev A, 18 March 1988.

1. **GAPP Programming** -- producing CT and MA code and merging into a microcode stream with two instruction streams [CT and MA] in three phases [shift, take over and main algorithm] that may partially overlap.

2. **SPDS Programming** -- software to control low level hardware associated with GAPP control:

   a. Control Store -- 32K 48 bit microcode storage

   b. Program Sequencer -- 16 bit pointer to next GAPP microcode word

   c. IO buffers -- four 64K 16 bit RAMs [ping-pong pairs on input & output of CT]

   d. CRAM -- 256 by 1 bit data memory loadable to GAPP c registers

   e. Start Vector -- 16 bit register vectoring to start of GAPP program

   f. Global OR Capture -- 16 bit shift register to capture global wired OR

3. **Host Interface programming** -- mediate transactions between host and SPDS controller using PIO and DMA:

   a. IO Port Addressing -- eight contiguous ports reserved & decoded, five used

   b. Wait State Generator -- variable delay programmable for 1-7 clock times.

4. **Control Programming** -- software to command and read status and to transfer data via PIO and DMA:

   a. Command Register -- 8 bit register storing current IO command

   b. Status Register -- 8 bit register reporting IO status

   c. Pointer Register -- 8 bit register denoting IO source/destination

   d. Data Register -- 16 bit register for PIO and DMA transfers

This section will discuss items 2-4 above, GAPP programming using the GAPP Algorithmic Language [GAL] is covered in the NCR SPDS documents.[3]

2.2 SPDS HARDWARE

The GAPP arrays are directly controlled by means of a 48 bit microcode word with the following interpretations:

---

3. "Software User's Manual", NCR Microelectronics - Fort Collins CO, Version 1.0, 1988

Bits	Symbol	Description
0-7	CADR0-7	address for fetching CRAM bit if bit 47 is 1
0-7	JMP0-7	LSBs of increment for jump commands if bit 47 is 0
8	JMP8	MSB of increment for jump commands
9-15	CTADR0-6	Corner-Turn RAM address
16-22	MGADR0-6	Main-array GAPP RAM address
23	CAPT_GO	Capture Global OR value in GO shift register
24-29	CTCMD0-5	Corner-turn command word
30-42	MGCMD0-12	Main-array GAPP command word
43-44	SEQ_CTL0-1	Main-array GAPP command word
45	BRK	Break from GAPP processing and wait for host
46	RTS	Return to start -- jam start vector to program address
47	CRAM/JMP*	CRAM address or jump interpretation of bits 0-7

2.2.1 GAPP Program Sequencing

Random Access Memory is provided for as many as 64K such microcode words in the Control Store [CS]. Microcode is downloaded from host to CS by DMA using the DMA address counter. In operation, the next CS address is determined in one of four ways:

1. **Count current address up by one** -- this is the normal case of program sequencing. It occurs when:

   - SEQ_CTL==00 [no jump]
   - BRK==0 [no break]
   - RTS==0 [no return to start]
   - CRAM/JMP*==## [don't care]

2. **Add contents of JMP0-8 [MCODE0-8] to current address** -- this is a forward jump by amount up to 512 locations. It occurs when:

   - SEQ_CTL==01||(SEQ_CTL==10&&GLOR==1)||(SEQ_CTL==11&&GLOR==0)
   - BRK==0
   - RTS==0
   - CRAM/JMP*==0

3. **Jam contents of start vector register to next address** -- the only way to decrease CS address and loop back to repeat microcode. It occurs when:

   - SEQ_CTL==##
   - BRK==0
   - RTS==1
   - CRAM/JMP*==#

4. **Retain current address** -- suspend SPDS operation when:

   - SEQ_CTL==##
   - BRK==1
   - RTS==#
   - CRAM/JMP*==#

These facilities allow limited program sequencing controls. Note that the only conditional branching possible at run time is based on global OR. The RTS facility is a primitive looping construct but it's all that is available.

2.2.2 GAPP Processing

Processing relies on GAPP commands and addresses. These are specifically **CTADR, MGADR, CTCMD** and **MGCMD** subfields of the microcode word. Full addressability for both CT and MA RAMs are provided and full command of MA processors. CT processor commands, however, are only a subset of the full GAPP repertoire, sufficient for shifting serial data and turning rows in parallel.

Except for the six CT command lines provided, all others are jumpered to zero. The CT subset is as follows:

CT	GAPP	Controls
01	01	Full cm [cm:=cm, cm:=ram, cm:=cms, cm:=0]
23	56	Partial ew [ew:=ew, ew:=ram, ew:=e, ew:=w]
45	BC	Full ram [read ram, ram:=cm, ram:=c, ram:=sm]

This subset provides full control of cm and RAM and partial control of ew. The ns and c registers are forced to zero on every cycle so behavior of the adder/subtracter is fully determined. These facilities make it easy to shift data using ew from West to East or from East to West and to ship it Northward on cm. Also, full RAM services are provided.

2.2.3 Global Data Facilities

There are two global data facilities, output to GAPP via CRAM and input from GAPP via global OR. Global OR state can be captured into a 16 stage shift register and then read by PIO or DMA into host. This capture occurs on command from the CAPT_GO bit in the microcode word. CRAM is a 256X1 bit memory whose content is forced into the c registers whenever the CRAM/JMP* bit is one. The address is taken from the first eight bits of the microcode word.

2.3 HOST INTERFACE PROGRAMMING

The hardware for these functions is packaged on a standard IBM-PC full length board which uses a 16 bit interface and so must plug into both connectors of an IBM-PC/AT or compatible bus. The UHI board decodes eight port addresses within the IO segment of the host and conditions bus signals for the SPDS controller. Four of these ports are decoded as four PIO pulses and forwarded to the SPDS controller for action. Additionally, bidirectional host data are transferred to the controller along with interrupt and acknowledge signals. The UHI is connected to the SPDS controller via a 60 wire cable as follows:

Signal	Wires	Description
UHI_DATA0-F		
	16	Data word (single-ended, bidirectional)
PIOWA*	2	Programmed I/O pulse A write (differential, out)
PIORA*	2	Programmed I/O pulse A read (differential, out)
PIOWB*	2	Programmed I/O pulse B write (differential, out)
PIORB*	2	Programmed I/O pulse B read (differential, out)
PIOWC*	2	Programmed I/O pulse C write (differential, out)
PIORC*	2	Programmed I/O pulse C read (differential, out)
PIOWD*	2	Programmed I/O pulse D write (differential, out)
PIORD*	2	Programmed I/O pulse D read (differential, out)
INT*	2	Interrupt request (differential, out)
XACK*	2	Interrupt acknowledge (differential, in)
	36	
RETURNS	24	
TOTAL	60	

Port addresses are selected by DIP switch settings on the board specifying the base port on an even quad-word; that is, with 000 as its three LSB. Then the base port gives PIO A, the next PIO B and so on. The fifth port is interpreted on the UHI for wait state generator control. PIORE* clears the wait state shift register and arms a programmable delay [1 to 7 clock cycles] beginning with the next XACK* from the SPDS controller board. Once the delay expires, the AT bus is signaled that SPDS is ready for PIO interrupt action via IO_CH_RDY [A10]. A value of zero disables IO_CH_RDY.

2.4 CONTROLLER PROGRAMMING

There are four key registers through which the control is exercised. They are the command, status, pointer and data registers. All are addressable via PIO through standard ports. The data register is also accessible via DMA and is interpreted as to source/destination according to the pointer register. Ports are used as follows:

Symbol	Address	Description
COMMAND	base	Command register, 1 byte, write and read
STATUS	base + 1	Status register, 1 byte, read only
POINT	base + 1	Pointing register, 1 byte, write only
DATA	base + 2	Data register, 1 word or 1 byte, read or write
UHI	base + 4	Universal host interface, wait states as above

2.4.1 Pointer Register

The pointer register is write-only and designates source or destination for subsequent read or write operations whether via PIO or DMA. Bit fields of this register are:

Bits	Symbol	Description
0-3	**DEVICE0-3**	Sink or source device pointer nybble
4	**MEM/IO***	Select DMA [1] or PIO [0] for succeeding operations
5-7	**CARD0-2**	Controller card ID selection [future options]

Interpretation of nybble DEVICE depends on MEM/IO*. For MEM/IO* = = 1 [DMA transfers], bits DEVICE2 and DEVICE3 are ignored and the remainder are interpreted as:

Manifest Symbol	Value	Description
AT_CSTORE	0x10	Usage: outp(POINT, AT_CSTORE)
AT_CRAM	0x11	Usage: outp(POINT, AT_CRAM)
AT_BUFFER_0	0x12	Usage: outp(POINT, AT_BUFFER_0)
AT_BUFFER_1	0x13	Usage: outp(POINT, AT_BUFFER_1)

For PIO transfers [MEM/IO* = = 0], bits 2 & 3 are used to modify interpretations of bits 0 & 1. For [DEVICE3,DEVICE2] = = 00, the lower two bits are interpreted as device data word transfers [read = = to host, write = = from host] as follows:

Bits	Direction	Description
00	read	Global OR capture shift register
00	write	DMA address counter
01	read	Corner turn output register [buffer 0]
01	write	Corner turn input register [buffer 0]
10	read	Corner turn output register [buffer 1]
10	write	Corner turn input register [buffer 1]
11	read	Next control store address
11	write	Control store start vector register

For [DEVICE3,DEVICE2] = = 01, the lower two bits are interpreted as pulse commands:

Bits	Direction	Description
00	write	Remove break interrupt
01	write	Toggle IO buffers
10	write	Clear IO buffer 0 counter
11	write	Clear IO buffer 1 counter

2.4.2 Command Register

The command register is one byte long and can be read or written by the host. It provides control commands and static modes of execution, clocking and various SPDS facilities. Command register bits are interpreted as follows:

Bits	Symbol	Description
0-1	**SEQ_CMD0-1**	Decoded into four (4) control store sequencing commands
		00 = = RESUME [continue execution after pause]
		01 = = ARM
		10 = = HALT
		11 = = RESET
2	**INT_EN**	Interrupts enabled -- halt/signal when global OR capture is full
3-4	**CLK_MODE0-1**	Decoded into 4 clock mode commands
		00 = = CONTINUOUS [run continuously]
		01 = = SINGLE LOOP [pause at first return to start]
		10 = = FAST [not implemented, same as 00]
		11 = = SINGLE STEP [pause after each cycle]
5	**ADR_INC_EN**	Enable stepping of DMA address counter
6	**INTERP**	Interpret control store microcode
7	**BUFR_IO**	Enable input/output buffers

2.4.3 Status Register

This is a one byte, read-only register reporting various conditions of the SPDS back to the host as follows:

Bits	Symbol	Description
0	**BUSY**	GAPP is running
1	**GLOR**	Global OR output
2	**BRKI**	GAPP is paused
3	**IO_PHASE**	Buffer set currently available
4-7	**GOCNT0-3**	Count number of captured global OR states

Section 3

Software

Because there are actually three different sets of hardware and they need to be programmed to coordinate their actions, the software structure gets somewhat complicated. In fact, the software of GACE is written in three different programming languages -- C, 80?86 Assembler and GAPP Algorithmic Language [GAL]. This last was developed by NCR for programming GAPP chips and a compiler was furnished by NCR along with the SPDS. In this section, we will outline this software structure and functional features of significance.

Five layers of software were developed and integrated to form GACE. These layers have considerable independence and data isolation from each other. They are:

1. Classifier system -- bucket brigade, bidding competition, genetic algorithms, etc. plus environment [entirely in C, analogous with an application program]
2. Function library -- C functions providing various special services for GACE
3. Kernel -- C functions to operate SPDS using small GAL programs [analogous with an operating system kernel]
4. Basic Input/Output Services -- C and Assembler functions providing basic input/output services [BIOS] for the kernel.
5. Driver -- device driver in Assembler providing hardware interface for BIOS via MS-DOS interrupt services.

In addition, two stand-alone utilities were developed. The first, **cgal** provides complete command line control of the GAL compiler. This allows easy compilation of GAL language programs directly in MS-DOS, without using the NCR user interface for SPDS. In addition, **cgal** applies the C compiler preprocessor to permit use of macros, included files and manifest constants in GAL sources. This greatly enhances readability of these sources and allows configured control of GAL parameters through "make" productions and another program that creates an include file. The **cgal** program functions much like the compile control programs [cc or cl] for C compilers. The various options permit compiling, linking and merging code for CT, take over and MA processing.

The second stand-alone utility is **stripbin** which takes the final output file from the GAL compiler and strips size, address and CRAM data. This output file, by convention, has extension 'bin'. The stripped file contains only microcode words in a format suitable for direct download to control store. The extension given its output by **stripbin** is 'csi' for Control Store Image.

3.1 CLASSIFIER SYSTEM

The following functions make up the classifier system:

File gace.c

init_cfs gets system parameters via **config**; allocates and initializes storage for classifiers, matches and message lists; initializes hardware and environment; downloads classifiers.

classify	run specified number of message matching cycles
bucket brigade	make payments from successful bidder to its suppliers
discover	select two parents and two victims, invoke crossover
cover_detector	generate detector matching rule from strong one, replace weak victim
main	loop for specified number of cycles, display results
i_act	action for command line switch option **i**, set input file root name
usage	tell user command line protocol

File gaceutil.c

load_classifiers	read and parse given file into classifier array
shuffle	randomly shuffle an array of pointers
delete_message	remove message from message list by pointer swap
strength_incr	step the strength of given classifier up or down
head_tax	apply accrued head tax for number of cycles since given classifier was last updated
sum_fitness	compute cumulative strength [fit_sum] and cumulative weakness [weak-sum] arrays for use in roulette selections.
roulette	select random element based on roulette model using binary search on fit_sum or weak_sum
enlarge_match_buffer	increase match buffer memory allocation
address	build address structure from dimensional pieces
print_bits	print bit string represented by an unsigned integer
print_trits	print trit values for one condition [pair of unsigned integers]
read_trits	read trit values for one condition from input file
hist	return string of asterisks representing unsigned integer in "thermometer" fashion for histogram displays
crossover	crossover two given parents, compute two children, replace two given victims
mutate	mutate random trits of a classifier

File fsw.c: Finite State World Environment

init_env	initialize finite state world environment
msg_input	put detector messages on classifier system message list
msg_output	scan message list for effector messages, activate any present

3.2 LIBRARY

The following functions are provided in library **isis.lib** for general use:

adapter	determine and report type of adapter and text parameters currently in use on host machine. Definitions and prototypes in file **adapter.h**
curdir	determine and report current directory
findsort	find files matching wild-card and search criteria and return a sorted list to caller
fixpath	standardize MS-DOS path and return to caller [for use by **findsort**]
getline	read line of arbitrary length from file. Dynamic allocation and reallocation used
getopt	read and parse options from command line arguments to program invocation. Perform caller specified actions for each
setopt	save command line arguments for use by **getopt**
isis.h	include file for this function library
msrand	minimal standard random number generator. Extensively tested
smsrand	set seed for **msrand**
parity	returns parity [number of ones] for its argument
simil	returns similarity index [0-100] for two input strings
getcommon	find length of greatest common substring [for use by **simil**]
stradd	allocate memory and form new string by concatenating two inputs
strdate	convert file modification date/time to a string for printing

3.3 KERNEL

The kernel is made up of both C functions and GAL beads running on GAPP under control of kernel C functions. There are seven C functions available in two versions. One set runs beads on the GAPP to provide parallel operations. It is contained in library **kernel.lib**. The other emulates GAPP operation on the host. These serial emulations are contained in library **s_kernel.lib**. The latter is slow but allows complete separation of classifier system/environment software from lower level software permitting independent development, debugging and testing. The seven C functions are:

File kernel.c or s_kernel.c

init_hdwr	sets up GACE, SPDS and bead tokens
reset	clears classifier condition match and message indices
match_vs	compare message against all classifier conditions
fire	set classifier rule fired flag based on condition matches

more	check if any more fired classifiers remain in GAPP
next	retrieve data on next fired classifier
load_cf	store modified classifier into GAPP data base

Beads

bits_b0.gal	transfers least significant byte of bits portion of condition trit
bits_b1.gal	transfers second least significant byte of bits portion of condition trit
fire.gal	sets fire flag for odd processor of classifiers based on both condition match flags and condition type
lbs_b0.gal	transfers least significant byte of lbs portion of condition trit
lbs_b1.gal	transfers second least significant byte of lbs portion of condition trit
load_cf.gal	transfers CRAM data previously loaded with classifier data to specific address in GAPP main array
load_col.gal	transfers column addresses from input area
load_row.gal	transfers row addresses from input area
madownup.gal	simultaneous download and upload of data bytes from I/O buffers through CT into/out of MA. Includes three instruction streams to be merged into two
match_vs.gal	compare CRAM word previously loaded with all condition trits
more.gal	determine if any fired classifiers remain, capture message indices if any, apply unique mark
next.gal	capture address of uniquely marked classifier
reset.gal	clear condition match bits and message index counters
shift.gal	test program to shift data through corner turn array
xfer.gal	test program to transfer main array input data directly to output area

In addition, there is a stand-alone utility called **mkgaldef**. This program reads GACE and SPDS configuration data from the configuration file, the MS-DOS environment or from defaults. It uses these data to create an include file called **galdef.h** which is included in all beads. This GAL definition header handles layout of main array GAPP memory and defines all GAL variables.

3.4 BIOS

The BIOS are made up of a number of functions incorporated in library **bios.lib** and written in both C and assembler. The functions [in C unless otherwise noted] are:

do_bead	host function to run a bead previously loaded at the given address in GAPP control store [assembler]

int_pack package of functions for handling 80?86 interrupts:

- int_capture - substitute vector
- int_vector - report vector
- int_release - restore vector
- int_kill - release all vectors
- int_test - test interrupt flag
- int_set - set interrupt flag
- int_clear - clear interrupt flag

mc_pack package of functions for handling 48 bit SPDS microcode words:

- mc_scan - disassemble microcode word
- mc_load - assemble microcode word

mess_load package of functions [assembler] for transfers to CRAM:

- mess_down - message to CRAM
- data_down - arbitrary data to CRAM

spdsconf read configuration data from MS-DOS environment, configuration file or defaults and set data structure

spdspack package of functions providing basic run-time interrupt services interfacing between kernel and driver:

- binify - set driver data mode to binary
- set_spds - configure driver
- spds_init - open and initialize driver
- data_down - download data to selected I/O buffer
- data_up - upload data, debugging only
- bead_down - download bead to control store
- bead_up - upload bead, debugging only
- put_bead - find, tokenize and download bead
- run_bead - check token and do_bead
- get_bead - bead_up per token, debugging only
- set_address - load GAPP addresses in main array RAM
- set_conds - load classifier conditions in main array RAM
- clear_map - reset token map
- gace_init - initialize SPDS and GACE

ui_pack user interrupt package:

- cc_int - signal control-C struck by user (DOS abort)
- cb_int - control-break interrupt service routine (BIOS abort)
- ui_install - install DOS and BIOS abort alternates
- ui_release - restore default DOS and BIOS aborts

3.5 DRIVER

A standardized device driver has been written in 80?86 assembler for the SPDS device. It meets all interface specifications for MS-DOS [versions 2.1 through 4.0], OS-2 [all versions] and Unix [version 7, System 3, System 5] device drivers. Only functions 0 through 12 are processed and, of those, all but 0 [initialize], 4 [input], 8 [output] and 12 [IOCTL output] immediately return either as not applicable or not implemented. The driver is of character type but with DMA transfers; block device specific commands are not applicable. Separate strategy and interrupt routines are provided as per the standards and these routines are vectored within the standard header.

- Command 12, [Input/Output control - output] is used to configure the device driver to its particular SPDS hardware. Port addresses and DMA delay values are transmitted from the user's process, via the operating system in a packet which is intercepted by the driver, never appearing in SPDS hardware. Any attempt to use the driver before configuring it results in a "device not ready" message from the operating system to the calling process.

- Command 0, [initialize] is invoked by the operating system kernel during boot and allows the driver to adjust system break [sbrk] for post-installation sizing.

- Commands 4 and 8 [normal input/output] are invoked through normal operating system kernel interrupts and/or operating system BIOS interrupts. These, in turn, are invoked via standard compiler or assembler calls from any of the various host languages.

Section 4

Testing

4.1 FINITE STATE WORLD ENVIRONMENT

The GACE parallel classifier system was run using the Finite State World (FSW) environment defined by Dr. Riolo. The task is to maneuver through a state space to obtain payoff by visiting high-value states as often as possible. The FSW environment[4] illustrated in Figure 1 shows the particular FSW (known as the GREF world) used in the runs discussed in this report. The GREF world has 16 states, numbered 0 through 15. From each state, there are transitions to a next state based on four input values which the classifier system may code into messages. In this Figure, these values are coded as colors. FSW classifiers are interpreted as rules like:

> If the message list has a message from the environment that the current state is 3, and there is an internally produced message with the form 10##101110##000#, then produce a message calling for action BLUE.

In the GREF world, payoff is received only when one of the states 12-15 is visited; positive payoff is received only in state 13. Hence, the classifier system's task reduces to visiting state 13 as often as possible. After one of 12-15 is visited, the world resets randomly into one of 0-3. In order to get to state 13 as often as possible (once every four steps), the classifier system must maintain and invoke rules to get from any of 0-3 to 13 three steps later. As a performance measure, total payoff is summed over each 100 step interval; maximum payoff per interval is 25000, minimum is -25, random performance gets an average of 3750.

4.1.1 Cases Run

The classifier system was run using three rule population sizes spanning two orders of magnitude, from the smallest reasonable population of 50 classifiers to the maximum possible of 5184. Five different methods of selecting an initial population were used, as itemized below, for a total of 15 test cases.

Data set 0 Rules generated randomly to match two environment input messages and produce an effector-activating message.

Data set 1 Random "one-condition" rules; both conditions are identical and match some random environment state, producing a random effector-activating message.

Data set 2 Six optimum rules seeded into data set 0. Perfect performance in the GREF world requires a minimum of six rules.

Data set 3 The six optimum rules, hooked into chains to allow the bucket brigade to pass back payoff to the "stage setting" rules.

Data set 4 A "complete" set of rules, in a population filled out with random rules. The rule set is complete in that all rules (individuals and short chains) which could possibly make sense are included.

---

4. Riolo, Rick L., "CFS-C/FSW1: An Implementation of the CFS-C Classifier System in a Task-Domain that Involves Learning to Traverse a Finite State World," Logic of Computers Group, Division of Computer Science and Engineering, University of Michigan, 1988.

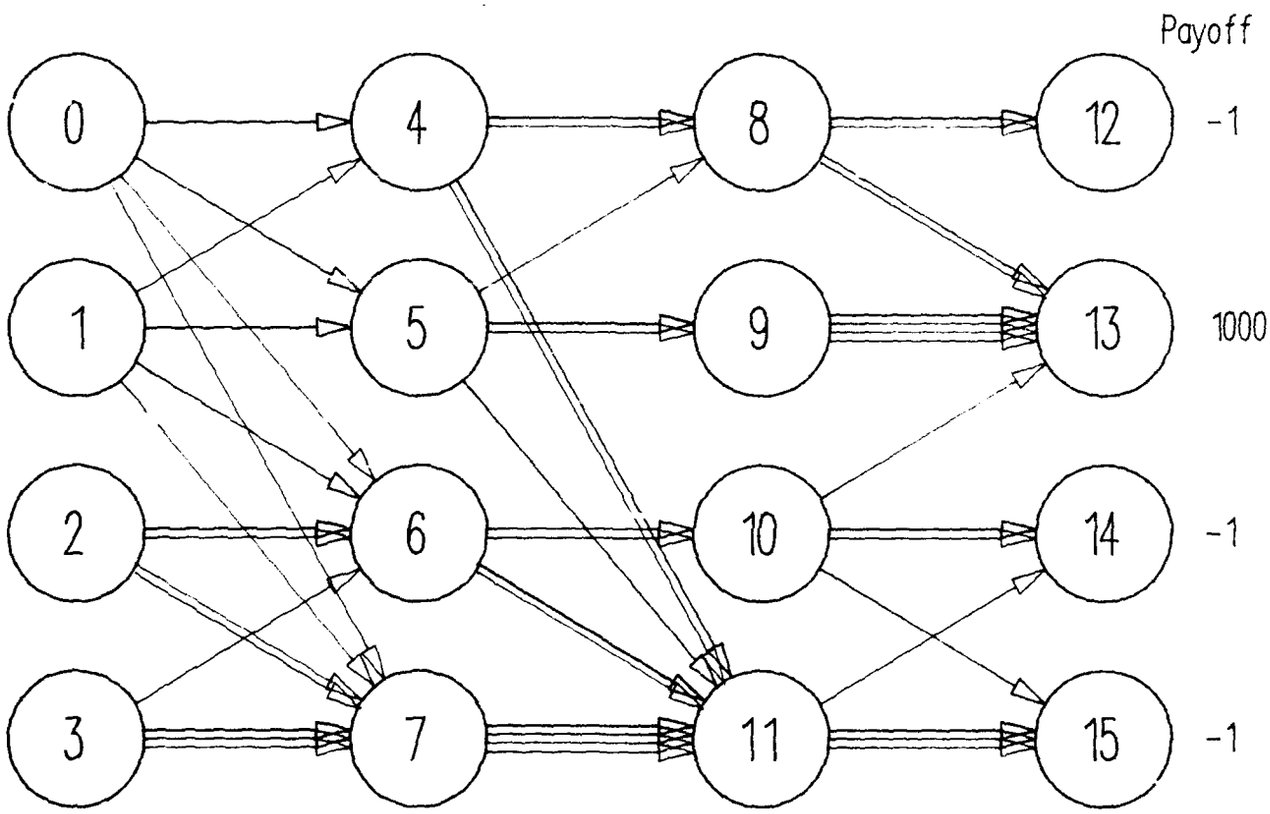

Figure 1: GREF World

4.2 RESULTS

The cases show that classifier rules can successfully navigate through the environment to get payoff, that the bucket brigade can differentiate between good and bad rules, that the discovery operators can change the performance of the system (though not necessarily in a beneficial manner), and that population size makes a big difference in qualitative performance. This last point leads to a number of areas for future exploration using the GACE.

4.2.1 Case By Case Analysis

Figures 2 through 6 display the results of the runs for each data set. Graphs for data set 0 show that the classifier system fails to achieve performance much above random for any population size. Average classifier strength quickly falls to nothing, since few rewards are received. The GACE classifier system leaves out many of the complex mechanisms previously developed to handle the very difficult task of building good rule chains out of random individuals--it was not expected to succeed given data set 0.

A bit of knowledge about the FSW task is included in data set 1. The environment produces exactly one message at each time step, reporting the current state. Rules in this set have both conditions the same, so the classifier responds to a single environment message. The bucket brigade is able to effectively evaluate these random "one-condition" rules when population size is small, and performance is better than random for a population of 50. The classifiers maintain their strength since rewards come more frequently. A significant percentage of data set 1 classifiers fire on each time step, and each good rule is likely to have several copies in the population. Since message list size is limited, only a few of these copies are allowed to fire and receive reward at one time. This limited distribution of reward prevents any one copy from maintaining its strength in the long run, and performance suffers for population 500. With a population of 5184, so many classifiers fire that their reports exceed the program's data space, preventing this case from running.

Data set 2 was included to see if the classifier system could build the chains needed to obtain maximum reward, given a few good individual rules as building blocks. The answer, in this case, was no. Of the six good rules, only one receives reward directly from the environment. The other five get no reward until they get hooked into chains to the rewarded rule. As it turned out, the pressures of taxes and the bucket brigade killed off the good rules before they could be made into chains. The rule receiving reward directly was able to survive but not proliferate after the stage-setting rules to make it applicable were gone. In the end, performance was not much above random.

The only difference between data sets 2 and 3 is that the six good rules are modified to implement chains, allowing the bucket brigade to provide reward to all six. In this case the larger populations are able to keep and replicate the chains to obtain virtually perfect performance throughout the run. However, the dilution of payoff discussed wrt data set 1 prevents the larger populations from maintaining their strength over time. Performance is maintained even after average strength reaches a minimal level because multiple copies of the chains are well established before strength fades. A population of 50, is not big enough to succeed with set 3. Due to the stochastic nature of the replacement algorithm, a particular classifier is more likely to be replaced in a small population; one or more of the necessary classifiers becomes a victim before it is able to establish its value.

Several interesting effects are demonstrated by data set 4. All three population sizes were able to evaluate rules to find the good ones, and were able to keep and build chains to get very good payoff. The population 50 run was not able to keep all the chains to get perfect payoff, but it was able to maintain classifier strengths. Population 500 obtained excellent payoff although most of its strength eventually drained away. Population 5184 got very good payoff, but lost its strength and adaptability before reaching the same level as the medium size population.

4.2.2 Advantages of Large Populations

- Large numbers of rules can be comparatively evaluated simultaneously.
- Individual rules are more likely to survive while being evaluated.
- More copies of good rules results in more stable behavior.
- Real world problems will require large rule sets.

4.2.3 Advantages of Small Populations

- Fewer cycles are required to evaluate rule sets.
- Good parameter settings are known from previous research.
- Easier to analyze contents of rule set.
- Serial machines may be used.

4.3 CONCLUSIONS & RECOMMENDATIONS

The GACE parallel classifier system is up and running. Message matching, rule firing, bucket brigade, and genetic algorithms are all operating correctly, and have been applied with some success to the Finite State World.

The basic mechanisms of classifier systems have been developed and refined using populations of at most a few hundred rules. Some revisions are necessary to get stable learning with thousands of rules. The revised classifier mechanisms will not necessarily be any more complicated, but they must exhibit better economic balance to resolve the dilution of bucket brigade payoff and the domination of rule sets by many copies of a few rules. The GACE has uncovered a number of problems, but provides an excellent research tool for finding a revised classifier system which solves them.

Research and development for GACE applications is ready to proceed in three directions:

1. The current machine's computational power should be used to systematically research potential classifier mechanisms. A better economic analogy for credit assignment is needed to replace the simple bucket brigade. The relationship between population size, task complexity, and rule discovery rates needs to be explored. The goal is to build a classifier system which uses its large population to perform well on a complex task in a changing environment.

2. The hardware used to implement the GACE is, at best, not optimized for classifier system operation. It does, however, demonstrate feasibility and provide design input for a next generation machine dividing tasks more efficiently between GACE and host. Development should begin now to take advantage of the results of the above classifier research.

3. The system architecture and low-level software developed for the GACE appear to have applicability beyond the field of classifier systems. Other fine-grained parallel systems can be programmed with relative ease to run on economical GAPP hardware. Work is needed here to demonstrate expanded applicability for the next generation machine.

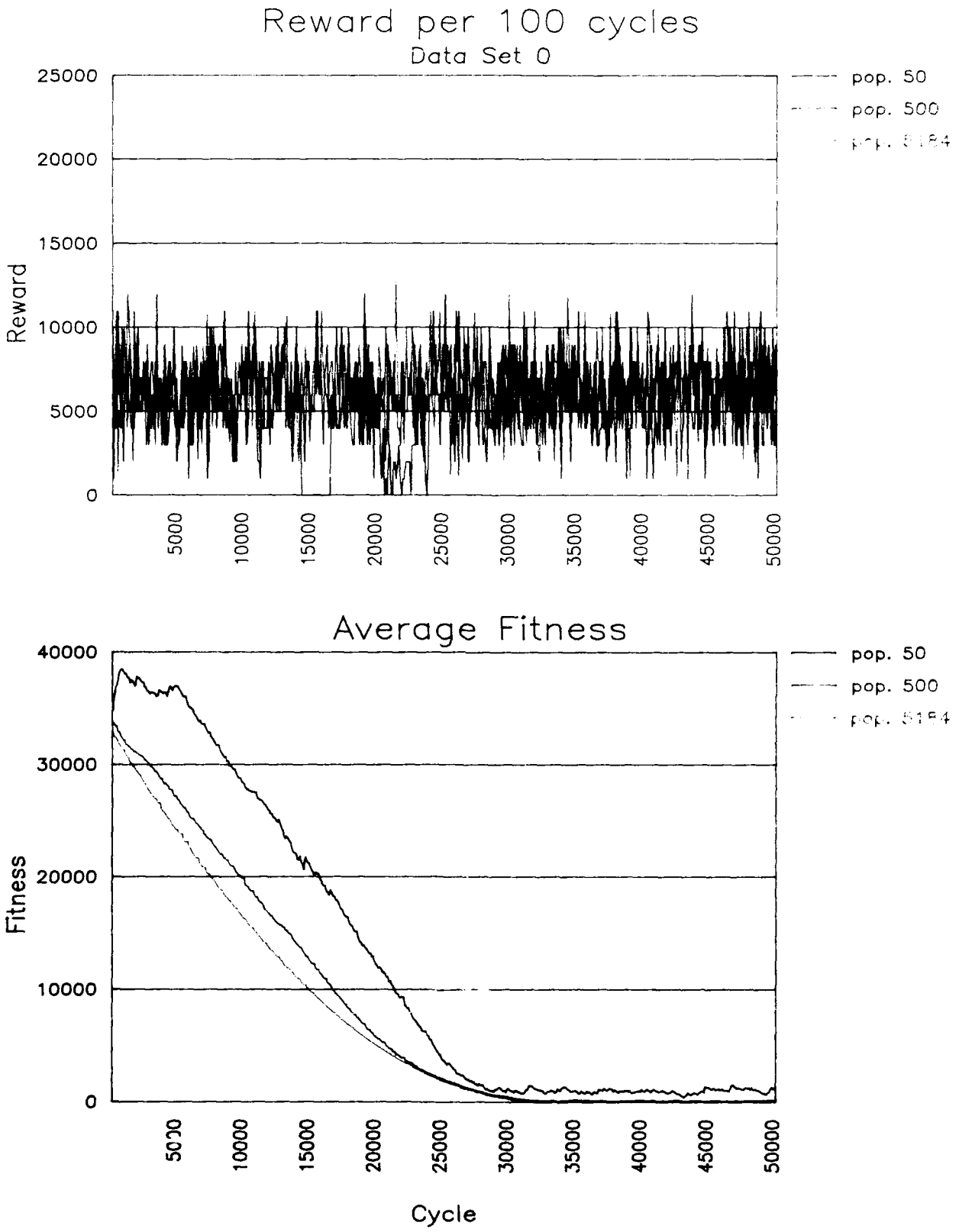

Figure 2: Data Set 0

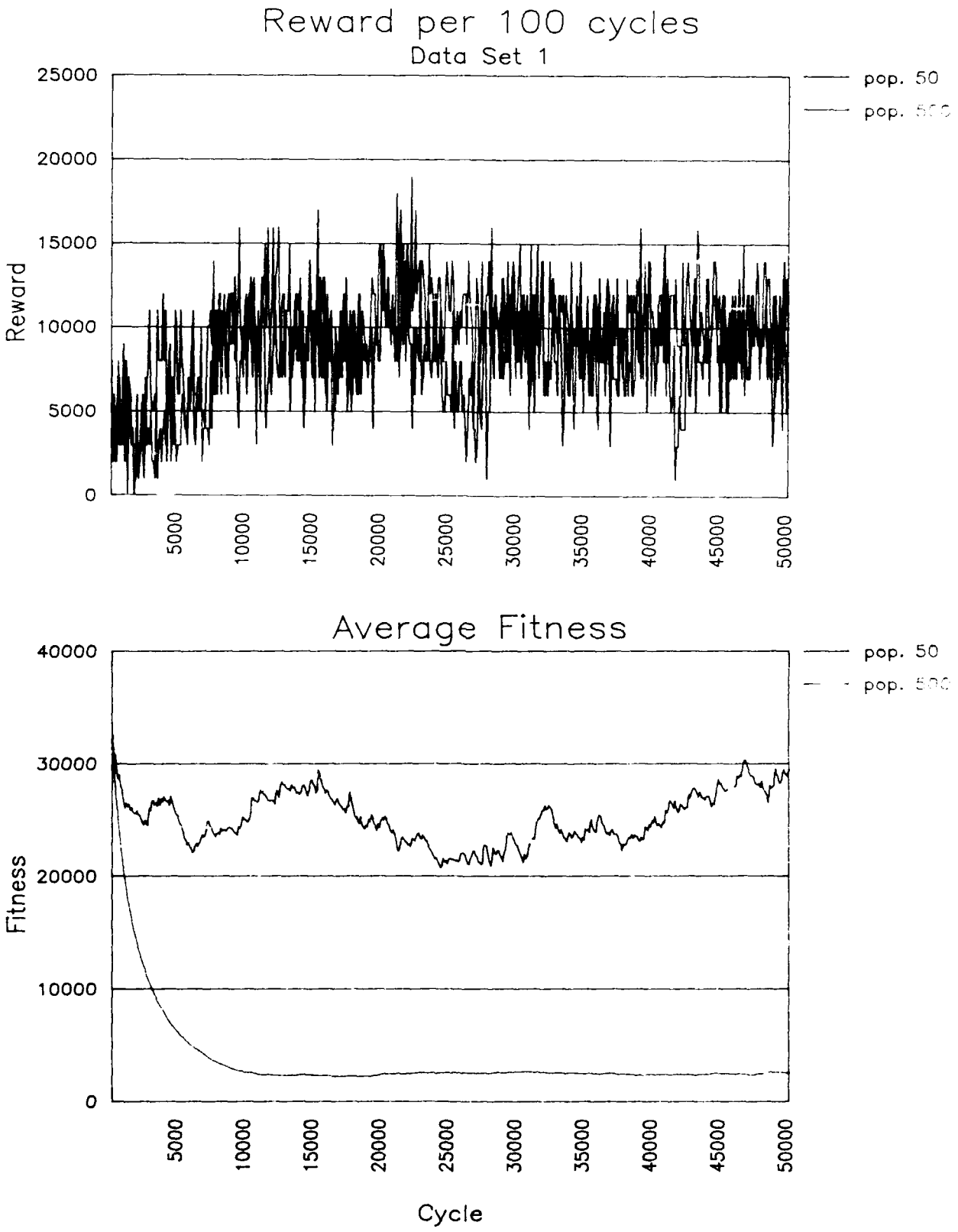

Figure 3: Data Set 1

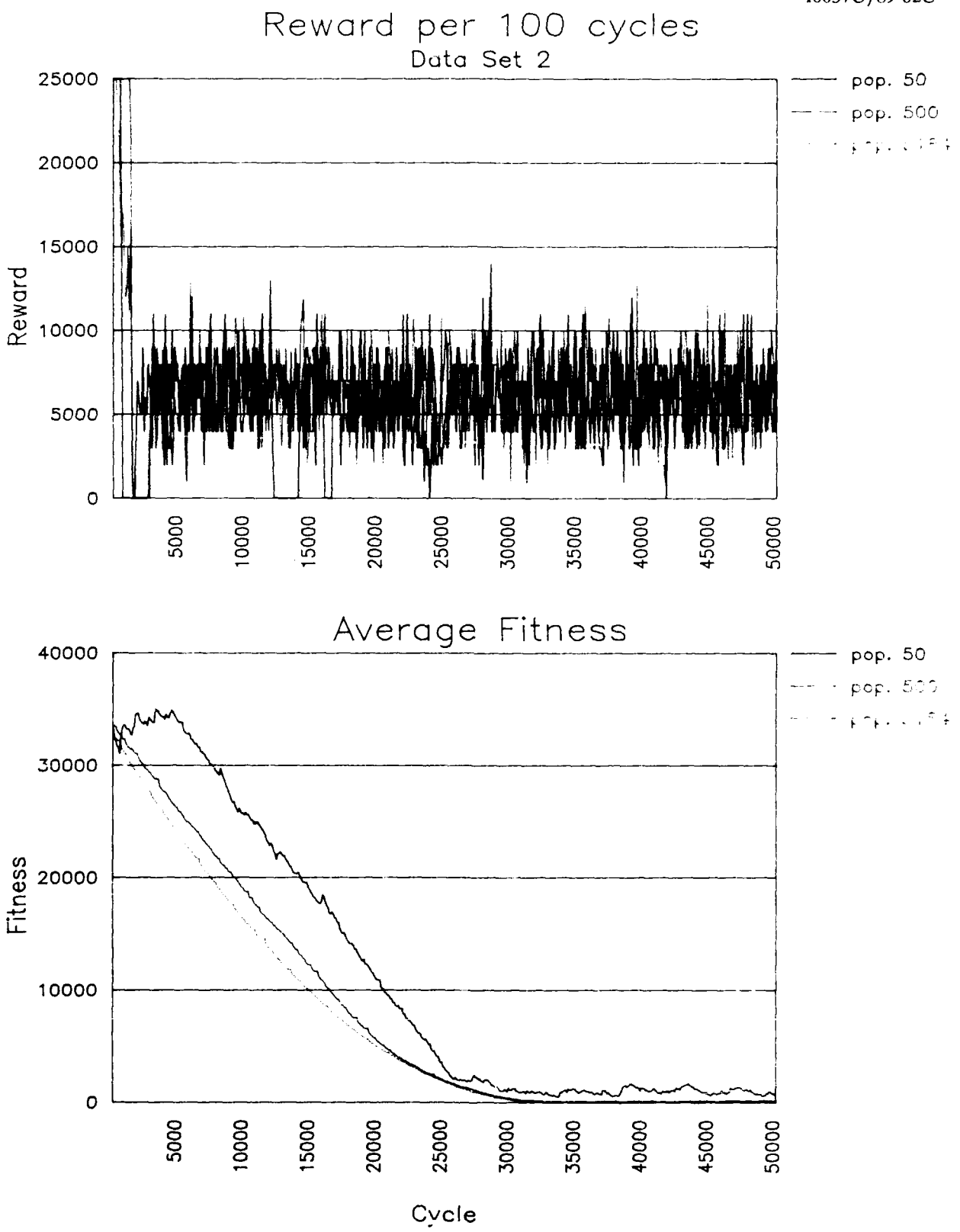

Figure 4: Data Set 2

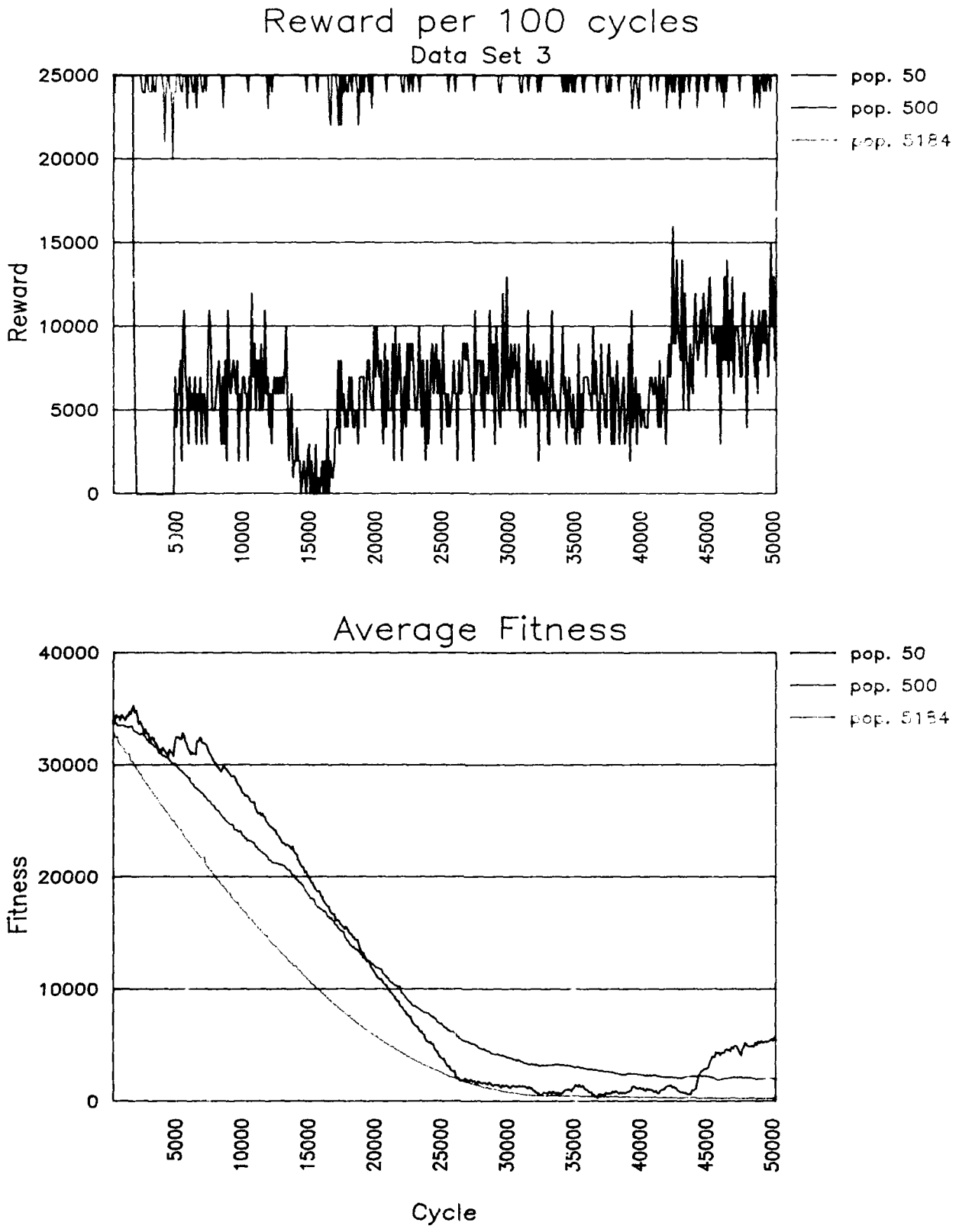

Figure 5: Data Set 3

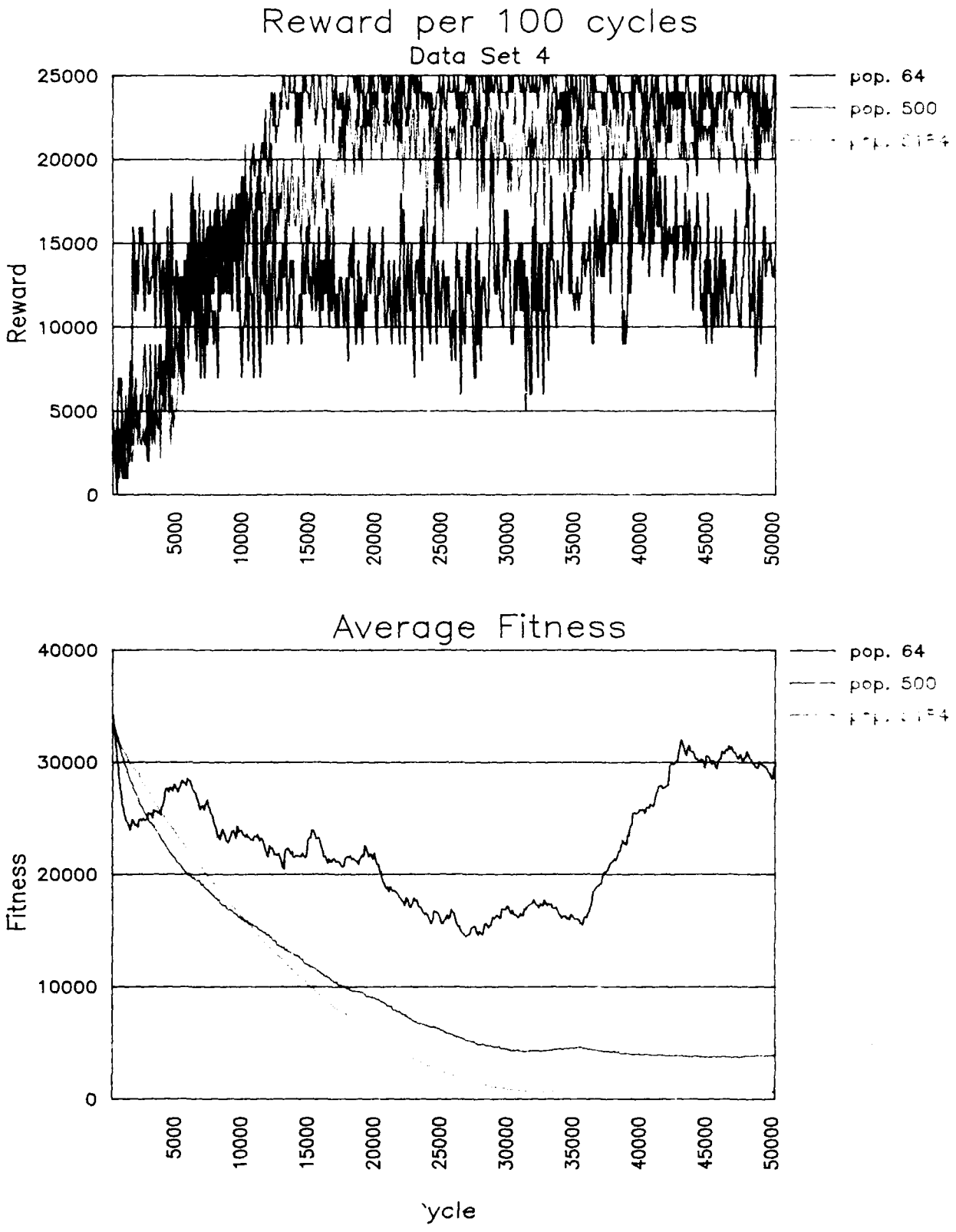

Figure 6: Data Set 4